Analysis of MapReduce Model on Big Data Processing within Cloud Computing

Marcellinus Ferdinand Suciadi[1]

***Abstrak*— Pada masa kini, komputasi awan menjadi tren pada pemrosesan data dalam volume besar. Google menciptakan model MapReduce untuk menyederhanakan komputasi kompleks yang biasanya menyertai pemrosesan data bervolume besar, dengan membagi-bagi data menjadi pasangan kunci/nilai, yang kemudian dapat diproses secara paralel, biasanya dalam jaringan, untuk kemudian digabungkan kembali menjadi hasil akhir. Walaupun demikian, model MapReduce memiliki beberapa keterbatasan. Peneliti telah berusaha mengembangkan model MapReduce, menghasilkan beberapa model terbaru, seperti model Mantri, Camdoop, Sudo, dan Nectar. Tiap model mengeksploitasi berbagai karakteristik dari model MapReduce secara unik untuk menghasilkan peningkatan kinerja pada kasus dan dengan cara tertentu. Walaupun demikian, tantangan dan peningkatan masih dapat ditemukan pada model-model ini, yang membuka berbagai kemungkinan baru untuk area penelitian.**

***Kata Kunci*: komputasi awan, MapReduce, pemrosesan data, model terdistribusi**

***Abstract*— Nowadays cloud computing is becoming a trend on big data processing. Google created MapReduce model to simplify the complex computation of big data processing by configuring and splitting the data into key/values pair to be processed in parallel, usually within a network of computers, then merge the results. However, MapReduce model has its limitations. Researchers have been trying to improve the model resulting in some newer models, such as Mantri, Camdoop, Sudo, and Nectar model. Each model exploits the different characteristics of MapReduce model to create improvements in different way and cases. Challenges and improvements still remain within these enhanced models, which open new possibilities on area of research.**

Keywords: cloud computing, MapReduce, data processing, distributed model

I. INTRODUCTION

Most of the popular applications in the world of today involve operations on large amounts of data resulting in a huge demand for cloud computing on the servers. The concept of cloud computing addresses the efficient usage of distributed resources coupled with parallel computing techniques to scale up development and deployment on a fail-safe infrastructure [12]. Such complex computing has however been simplified greatly by Google's MapReduce, which is a data processing tool that allows processing huge volume of data over clusters of low-end computing nodes. The design itself is an abstraction that enables simple computations to be expressed whilst automatic distributed computing and fault tolerance are handled in the backend library [1]. However, the MapReduce has limitations in its framework where recent approaches have exposed and presented new designs to overcome such limitations. In this article, an overview and discussion of the recent major approaches aimed at enhancing the MapReduce will be presented. The rest of this article is organized as follows. Section 2 describes the MapReduce framework and some of the key features involved. Section 3 presents the details of recent approaches for the improvements and extensions to MapReduce. Section 4 discusses and overviews the key techniques introduced. Section 5 explores open issues and challenges. Finally, Section 6 concludes this article.

II. MAP-REDUCE

Programmers at Google have created and implemented a distributed model called MapReduce, a model which consists of two parts: a map function to process key/value pairs to create intermediate key/values, and a reduce function to merge all intermediate key/values [1]. Let k_1 be set of keys and v_1 be set of values. The Map function takes pairs of k_1 and v_1 respectively to generate intermediate key, I, which consists of pairs of another keys, k_2, and values, v_2. This intermediate key is then passed as input for the Reduce function, which will merge all values to generate a smaller set of values.

A. *Implementation*

MapReduce computation can be used to count an URL access frequency, index a document, or even distribute a sort. Because an input data can be very large, MapReduce execution can be distributed. A user will start this model by splitting the input data into M chunks. These chunks are usually limited in size, typically 16 MB. Copies of this model are then executed on different machines called workers, processing M Map functions and R Reduce functions, controlled by a special copy of the program called *master*. To prevent faults, master will ping the workers occasionally. If a worker does not respond, the master will mark the task as failed and reassign a new worker, typically informing other workers that the task is

[1] *Dosen Jurusan Teknik Informatika Fakultas Teknik Universitas Surabaya, Jl. Raya Kalirungkut, Surabaya, Jawa Timur 60293, Indonesia (tlp: 031-298 1395; email:ferdi@staff.ubaya.ac.id)*

being rescheduled. Since the data from a finished Map task is stored on local disk of the failed worker, rendering them inaccessible, this task needs to be re-executed. The data from a finished Reduce task, in the other hand, is stored on a global storage system, so if a worker fails, this task does not need to be re-executed. However, if the master fails, the whole process will be halted and marked as failure. Users need to do a recovery task to continue the task, which can be brought from the last checkpoint before the master crashes.

B. Refinements

Some refinements can be made to this simple MapReduce model. If the Reduce function is commutative and associative, a Combine function can be executed firsthand before the Reduce function. A Combine function combines the partially data produced by Map functions and passes them to the Reduce function. The Combiner and Reduce function can be implemented with a single code, however a Combine function will produce an intermediate file to be sent as an input for the Reduce function, whereas a Reduce function will produce a final output file. Another refinement is the ability for a MapReduce model to detect faulty records and skip processing them in order to prevent crash. This is provided because sometimes there might be bugs in user-defined Map or Reduce functions that will make the system behave incorrectly or abruptly, halting the whole process, and sometimes it is impossible to correct the bug (for example when using third-party modules for Map/Reduce function). When there is a fault in the Map or Reduce function, a worker will send a signal to the master that a particular record causes the function to some errors. When the master has collected more than one of these signals, it can conclude that the record itself is faulty and should be skipped for the next execution of MapReduce task. This ability can also be used to intentionally skip some records.

III. IMPROVEMENTS AND EXTENSIONS TO MAPREDUCE

Despite the refinements made in MapReduce framework, some of implemented general mechanisms as described in the above section may have problems in certain cases and result in poor over performance. As such, recent approaches have come up with different strategies to optimize and enhance Map-Reduce.

A. Reining in the Outliers in MapReduce Clusters using Mantri

There are different phases in the jobs scheduled for data processing. More often than not, tasks in a particular phase may require the outputs of previous phases as inputs to complete the job. Hence, when certain tasks do take a longer than usual time to finish, the total time for the job taken will be lengthened greatly. Although MapReduce duplicates the remaining in-progress tasks when an operation is near completion to handle issue of stragglers, such a general mechanism is not ideal. The authors of Mantri argues that only acting at the end of a phase, opportunities to achieve lower job reduction time by dealing with outliers identified early while using fewer resources will be lost [13].

1) The Outlier Problem

The authors of Mantri [13] first understand the mechanics of outlier problem before drafting an optimized design. Their authors observe that duplicating high runtime tasks that have large of amount of data to process will not make them run faster hence leading to wasted resources. On the other hand, high runtime tasks that cannot be explained by data they process are likely due to resource contention or bad machines present and they may result in faster job completed time if scheduled to run on another location.

Also, the reduce phase will cause high crossrack traffic as the output of map tasks are distributed across the network of machines. Hence, when reduce tasks are simply placed on any machine with spare slots, it may lead to outliers due to the fact that a network location with many reduce tasks will likely have its downlink highly congested with reading of map task results operations.

Lastly, investigations show that the occurrence of recomputes due to straggling tasks is correlated with higher usage of resources. The subset of machines that triggers most of the recomputes is steady over days but varies over weeks, likely indicative of changing hotspots in data popularity or corruption in disks. Recomputation affects jobs disproportionately and they manifest in select faulty machines and during times of heavy resource usage [13].

2) Mantri Design

Based on the findings in the section above, Mantri is designed to act on the outliers identified early for higher efficiency of outliers handling while conserving additional resources used. However, there can be cases where remedy actions may result in longer job completion time or higher resources wasted should prediction of initial estimates of threshold be incorrect. Therefore, Mantri uses real-time progress reports through a Closed-loop to act optimistically by keep tracking of the cost as the probabilistic predictions go wrong.

A restart algorithm is written to perform intelligent restarting of outliers. The main idea is to check if an outlier that has a long runtime is due to the fact that it has a large amount of data to process or that it is slowed down due to its location. A task with real work will not be restarted. On the other hand, if a task lags because of reading data over a low-bandwidth path, it will be restarted only if a more advantageous network location becomes available or the task will be duplicated instead. As such, Mantri uses two variants of restart: killing a

running task then restarting it elsewhere and scheduling a duplicate task. By computing the following two variables using the task progress reports for each task, t_{rem}, the remaining time to finish, and t_{new}, the predicted completion time of a new copy of the task, Mantri will perform a restart only when the probability of success, $P(t_{new} < t_{rem})$ is high. If the remaining time to finish a task is so large that a restart would probably finish sooner instead, i.e $t_{rem} > E(t_{new}) + \Delta$, Mantri will proceed to kill and restart the task as such a scheme greatly shortens the job completion time without the need of additional slots. However, the queuing delay incurred by job scheduler before restarting a task can be pretty large. On the contrary, scheduling duplicates does not involve queuing and can achieve better performance when duplicate tasks end faster than the original. Nevertheless, duplicates require additional slots and computation resources which may result higher job completion time should there are outstanding tasks. Hence, duplicate is scheduled only when total amount of computation resource consumed decreases given that there are outstanding tasks and no slots is available, $P\left(t_{rem} > t_{new}\frac{c+1}{c}\right) >$. For stability sake, Mantri ensures no more than three copies of same time will be running concurrently. A task will not be reduplicated if a duplicate has already been launched for it recently and if a copy is slower than the second fastest copy of the task, it will be terminated to avoid wasting resources. However, towards the end of job where more slots are available, Mantri will schedule duplicates more aggressively than before.

As a rack with many reduce tasks will have its downlink congested leading to outliers, Mantri will consider both location of data sources and current utilization of network when placing a task. Instead of the solving a common and challenging central placement problem, Mantri uses a local algorithm which does not require updated network state information and centralized coordination. The key idea is that each job manager will allocate tasks in a manner where load on the network is minimized and self-interference among its tasks is avoided. From the size information of map outputs in each rack, two terms will be computed. First term is the ratio of outgoing traffic and available uplink bandwidth, $c_{2i-1} = \frac{d_u^i}{b_u^i}$ and second term is the ratio of incoming traffic and available downlink bandwidth, $c_{2i} = \frac{d_v^i}{b_d^i}$. The local algorithm then finds the optimal rack location where maximum data transfer time is minimized for each task by computing all placement permutations. For non-reduce phases, Mantri will use Cosmos policy of placing a task close to its data. Furthermore, Mantri computes the cost of moving data over low bandwidth links in t_{new} to avoid the case where copies are started at a location where it has little chance of finishing earlier thereby not wasting resources.

Mantri acts by an algorithm which replicates task output to mitigate the problem where costly recomputations will stall a job. Task output will be replicated early as Mantri weighs the cost of recomputation against the cost of replication. Essentially, the algorithm will consider the following three scenarios for replication: task output is very small and replication cost is negligible, tasks run on possibly bad machines and when cumulative cost of not replicating successive tasks is high. Mantri also controls the amount of data replicated to 10% of the data processed by the job through a token mechanism to avoid too much replication. On the other hand, pre-computation is carried out if Mantri estimates that a recomputation may likely cause future request for data to fail. Both probabilistic replication and pre-computation are employed to further enhance the efficiency of Mantri design.

B. In-Network Aggregation

Over the years, many researches have been conducted to improve the MapReduce model. One attempt is Camdoop[2], a framework that aims to optimize network traffics during the shuffling phase in MapReduce, when data are aggregated over servers. In typical MapReduce implementation, there are O(N2) flows of traffic data over participating machines. As [Costa, P. et al] show in their paper, there is a server link bottleneck, and increasing the bandwidth does not solve the problem entirely, especially for small networks. Instead, they reduce the amount of data transmitted in shuffle phase. A MapReduce-like system, called Camdoop, is run on a CamCube platform, that enables the system to reduce the number of Reduce tasks to be an expected number of outputs, rather than the usual number of intermediate pairs. CamCube uses a direct-connect topology to connect every server, as shown in Figure 1, thus no longer differentiate logical and physical network with the use of direct-connect topology and servers that are able to handle package forwarding. By exploiting these two features, Camdoop results best performance with help of custom routing and transport protocol, as well as in-network aggregation.

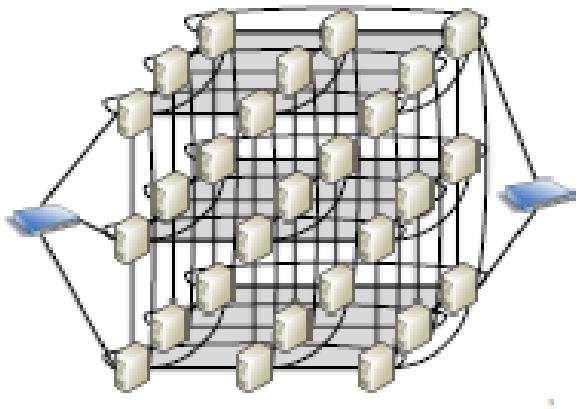

Figure 1. A CamCube topology

Camdoop works by splitting input data into chunks that are uniformly distributed, just like MapReduce model. Each server is constructed in a tree topology as in Figure 2, where the root will execute the Reduce task, spread amongst its leaves that will compute the key and value pairs before sending them back to the root to be aggregated and stored. Every job has a *jobID*, job description, user-defined Map and Reduce functions (or any other functions where required), and any deterministic MapReduce parameters such as *M* and *R*. The input data is split into chunks that have their own IDs, *chunkID*, which is a 160-bit identifier that is ensure a uniformly-distributed chunks. Map task is then run on servers to produce intermediate key values, just as in a typical MapReduce model. Each output has their own ID, *mapTaskID*, which is typically just the same as the *chunkID* of any input chunk processed by its task. When all intermediate key values have been written to disk, shuffle and Reduce task then commence.

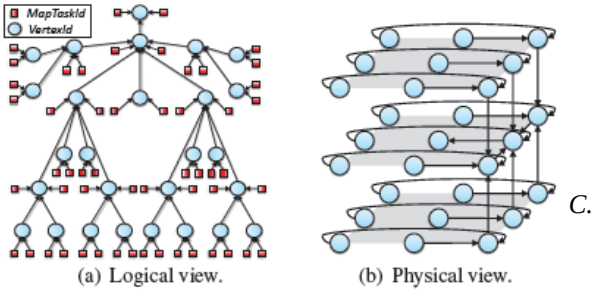

Figure 2. Tree topology in Camdoop

To implement the tree topology, each server has its own *vertexID*, which is a 160-bit identifier. The first k bits of this ID is used to uniquely identify every server on the 3D coordinate in the CamCube network, while the remaining bits are a hash value from a job ID. The server that runs a Reduce task has a *rootID* which is just its own *vertexID*. To determine its parent, each server implements a *getParent(rootID, id)* function, where *id* is either a *vertexID* or a *mapTaskID*. When *id* is a *mapTaskID*, this function takes the first *k* bits from the *ID* and returns a *vertexID* that generates the same coordinate. When id is a *vertexID*, this function returns another *vertexID* that is one level higher in the 3D space from the input *vertexID*. For a simple case where there is only one Reduce task and assuming that there are no failures, only one *vertexID* is mapped to a server.

To evenly distribute the workload for all trees, Camdoop creates six disjoint spanning trees with the same root, as shown in Figure 3. Using this topology, each physical link is used by one parent and one child in every direction, in effect of evenly-distributed workload. The only requirement to keep in mind is that intermediate keys need to remain in order and consistent for all stripes. However, this is not feasible when the Reduce tasks are very large in numbers. Instead, the authors suggest to create six disjoint spanning tree for each Reduce tasks. Assuming that there are no failures, each link is shared amongst *R* Reduce tasks.

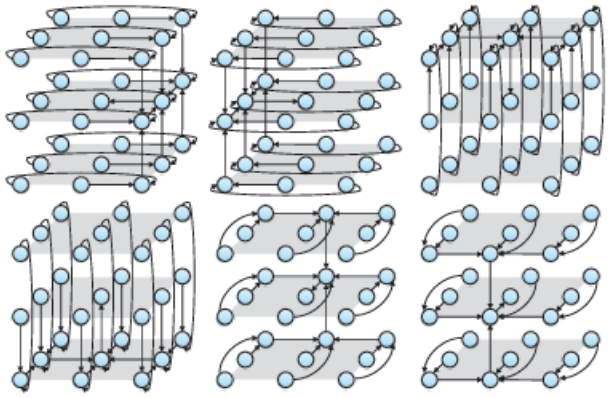

Figure 3. Six disjoint spanning trees

To address with failures, Camdoop recomputes the shortest path to a working server which is nearest to the failed server, reroutes the packages to the new path, and notifies all servers about the failure and the new path. However, it does not address an issue in MapReduce model when a server that stores intermediate values crashes, hence the task need to be restarted.

C. Data Shuffling Optimization

Another approach to optimize MapReduce model is Sudo model as proposed by Jiaxing Zhang et al [3]. Sudo exploits data shuffling stages that prepares data for parallel processing. For example, the intermediate key/value lists need to be sorted and resplit for distribution amongst servers before passed into the Reduce function, then be remerged. This process is known to be expensive in terms of disk I/O processing and network bandwidth since it involves all data. Sudo tries to learn the behavior of resulted data in each phases of MapReduce model to avoid data shuffling as possible.

TABEL I.
DATA PARTITION IN SUDO

Within-partition / Cross-partition	None	Contiguous	Sorted
None	AdHoc	-	LSorted
Partitioned	Disjoint	Clustered	PSorted
Ranged	-	-	GSorted

It has been observed that data partition has some properties between another partition and in-between the partition itself. Six configuration is found and can be seen in Table 1. The relationship between these properties is shown in Figure 4, where the topmost property is the strongest. Normally, data partition occurs in three steps: sorting records within a partition according to a key, repartition the records, and remerge the redistributed records based on a key. Not all property requires all three steps, as shown in Figure 5. A directed acyclic graph

(DAG) is used to model a data-parallel job, which consists of three type of vertices: data vertices that are responsible for input/output, compute vertices that do the computational phases (map, reduce, or merge), and shuffle vertices that do the data-shuffling. Sudo tries to optimize data shuffling by finding a valid execution plan with lowest cost for each job. The two optimizations are the use of functional properties of user-defined functions and redefinition of repartition function.

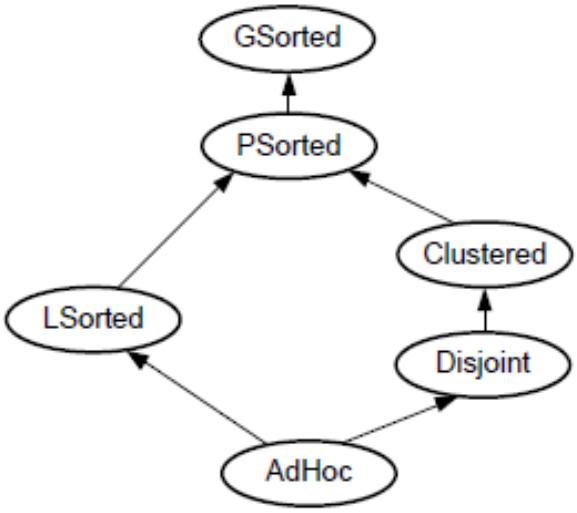

Figure 4. Relationships between data partition properties in Sudo

The authors of Sudo observe that in a traditional MapReduce model, user-defined functions are often regarded as "black boxes", which may not preserve the properties of data-partition of the input and user-required properties for the output. If it is known that the data-partition from a previous step already holds all expected properties, then data-shuffling is no longer necessary dan may be ignored. These user-defined functions, when constructed with some properties (functional properties), can be turned into "gray boxes" that eliminate the need of data-shuffling. To do this, Sudo executes additional tasks at the beginning of each job. The first is user-defined functions analysis to extract their properties and backward WP-analysis to determine the weakest precondition before each computational phase and weakest postcondition (hence the name of WP) after each data-shuffling phase. The second is a forward data-partition property propagation to create valid execution plans which have optimized data-shuffling. Finally, a plan with the lowest cost is then chosen.

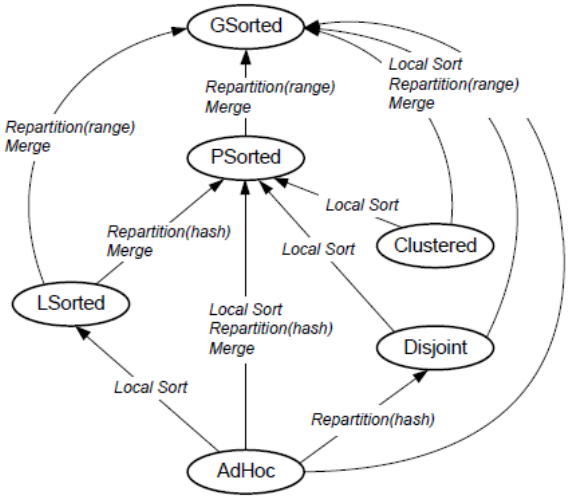

Figure 5. Data partitioning steps for each properties

According to the authors, a functional property defines the relationship between the inputs and outputs of a particular user-defined functions. Sudo is only interested in deterministic functions, where an output is created from an input of a record. After observations, there are three interesting functional properties. Let $f(x)$ be any deterministic function. Function $f(x)$ is said to be strictly-monotonic if and only if for any x_1 and x_2 in the input data, $x_1 < x_2$ always preserve the fact, either $f(x_1) < f(x_2)$ (strictly-increasing) or $f(x_1) > f(x_2)$ (strictly-decreasing). Likewise, function $f(x)$ is said to be monotonic if and only if for any x_1 and $x_2$in the input data, $x_1 < x_2$ means either $f(x_1) \leq f(x_2)$ (increasing) or $f(x_1) \geq f(x_2)$ (decreasing). Lastly, a function $f(x)$ is said to be one-to-one if and only if for any x_1 and x_2 in the input data, $x_1 \neq x_2$ means $f(x_1) \neq f(x_2)$. Sudo also defines a pass-through function f(x), which is a function that produces the same output as the input. These functional properties correspond to data-partition property in such a way shown in

Figure 6.

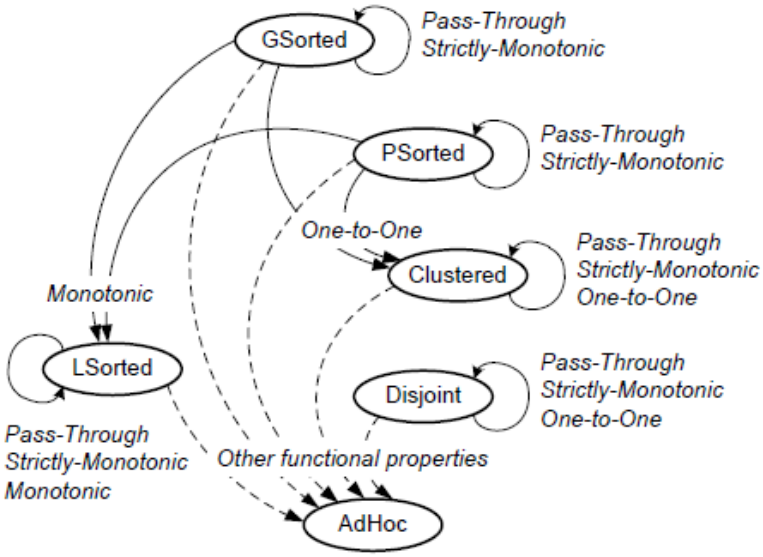

Figure 6. Correspondence between functional properties and data-partition properties

The second optimization Sudo proposes is the redefinition of repartition function. It is observed that sometimes user-defined functions do not preserve desired properties. Sudo allows redefinition of a partitioning key in order to maintain desired properties. Figure 7 shows an example on how such an optimization can be done. However, the authors point out that there are some side effects on this optimization. The repartitioning process is slightly more expensive and may result in larger number of records for the next mapper phase. This can be cured by a program slicing on the repartitioning function. The other side effect is the chance to get data skew. Therefore, before applying this optimization, a cost model is run to determine whether or not these side effects are tolerable.

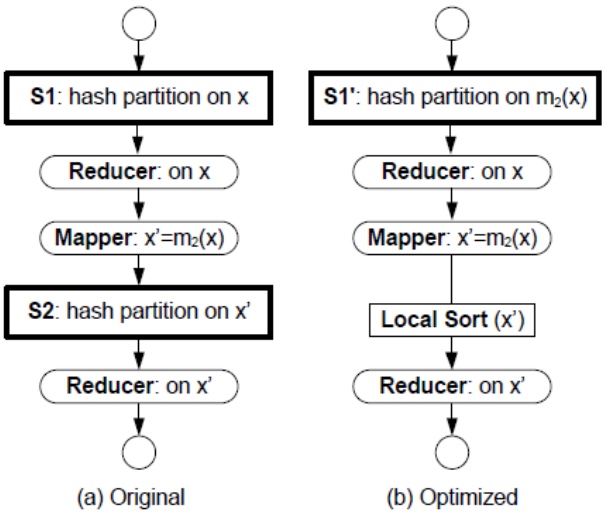

Figure 7. Example on partitioning key redefinition

D. Automatic Management of Data and Computation

MapReduce has greatly simplified the development of large-scale, parallel data processing applications. However due to a lack of efficient management of data and computation, large amounts of resources are wasted through redundant computations and mishandling of obsolete datasets. The authors of Nectar [14] present a system that manages the execution environment of a datacenter and is designed to address the aforementioned problems. Nectar implements a cache server and a garbage collector to effectively manage computation results and derived datasets, thereby providing advantages like efficient usage of resources through space utilization, reuse of shared sub-computations, incremental computations and ease of content management.

1) Nectar Architecture

The DryadLINQ programs are written to perform computations carried out in a Nectar-managed datacenter. Nectar can collect information of the program and the data dependencies as the set of functional operators in LINQ programs access datasets of .NET objects and transform the input datasets to new output datasets. Two classes of data are stored are described in the datacentre. Primary datasets are created once and accessed and derived datasets are the results from computations of primary or other derived datasets. It is important to note that Nectar can reproduce the derived dataset even when they are deleted automatically in the future, as Nectar keeps a mapping between a derived dataset and the program that creates it. Primary datasets however, are not deleted as they are referenced by conventional pathname and cannot leverage on the mapping property.

As shown in Figure 8, these DryadLINQ program begins as input and is handled by a program rewriter

inside a Nector Client-Side Library. The program rewriter will consult the cache server for cache hits in rewriting a more efficient program, which will be given back to DryadLINQ to be compiled into a Dryad Computation to be run in the cluster. Both input and output of DryadLINQ are stored as streams in TidyFS, an in-house distributed and fault tolerant file system.

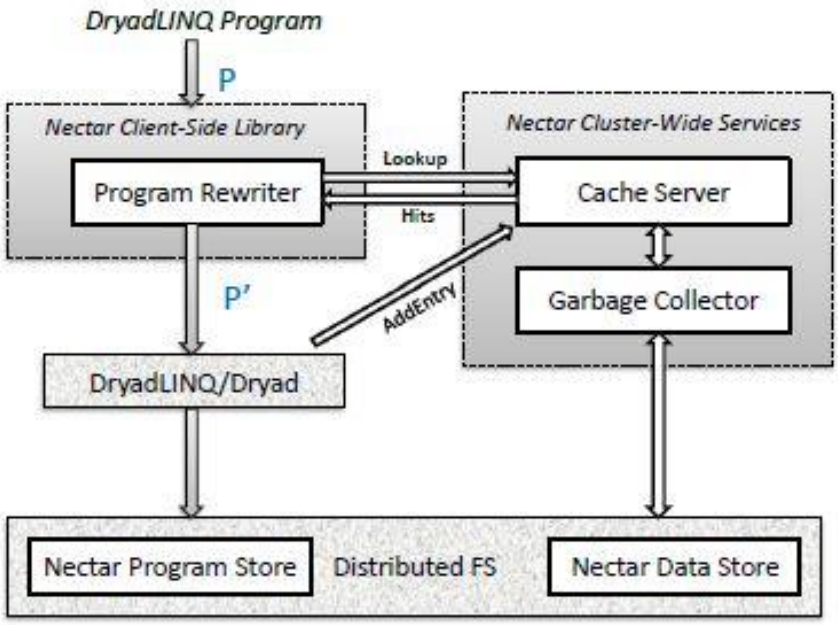

Figure 8. Overview of Nectar Architecture

The program store keeps all DryadLINQ programs that have ever executed successfully and the data store keeps all derived streams from the programs. A replacement policy is included in the cache server, where cache entries that have little value are deemed to be garbage and is deleted permanently by the garbage. Oh the other hand, programs in the program store will not be deleted as they are necessary for the recreation of derived datasets where required in the future.

2) Caching Computations

A cache entry records the result of executing a program on some given input and is of the form: $<FP_{PD}, FP_P, Result, Statistics, FP_{List}>$, where FP_{PD} is the combined fingerprint of the program and its input datasets, FP_P is the fingerprint of the program only, *Result* is the location of the output, and Statistics contains execution and usage information of this cache entry. The last field *FP*List contains a list of fingerprint pairs each representing the fingerprints of the first and last extents of an input dataset [14]. The finger of inputs is formed by combining the fingerprints of actual content in the dataset. However, computation of program fingerprint may be an issue as the program can contain user-defined functions that call into library code. As such, Nectar implements a static dependency analyzer to capture all dependencies of an expression and computes all the code necessary in a program to form the fingerprint. The statistics information in the cache entry will be used to find an optimal rewriting execution plan, the cache insertion and eviction policy.

In general, the process of rewriting programs to equivalent but more efficient one involves the identification of all sub-expressions of the expression and checking the cache server for cache hits on these sub-expressions. A set of equivalent expressions are written using the cache hits optimally using cost estimation to calculate maximum benefit.

Only cache hits on prefix sub-expressions on segments of the input dataset are taken care of as considering all possible sub-expression and subset of input dataset is not feasible. The rewriting algorithm involves simple recursive procedure, starting from obtaining all the possible hits from the cache server, *H*, on the largest prefix sub-expression which is the entire expression, *E*. If there is a hit on the entire input, the hit will be used for rewrite as it gives the most savings in terms of cumulative execution time, otherwise the best execution plan will be computed for *E* [14]. Performing this procedure through brute force search will not be efficient. Hence, Nectar stores fingerprints of the first and last extents of the input dataset in the cache entry and thereby computing *H* in linear time with the information.

Nectar also implements a cache insertion policy for decision making since it is not practical to cache every successful candidate for caching determined. The final result of a computation is always cached as it is considered free to obtain, while sub-expression candidates

are cached only if they are deemed to be useful. Number of lookups and runtime information from the execution of sub-expressions are used for making that caching decision based on a benefit function. At the same time, the cache insertion policy is adaptive to storage space pressure, where if space is available, candidates will be cached more easily.

3) Nectar Architecture

All the derived datasets will consume large amount of storage space if left alone. Due to the nature of TidyFS, the programmers are not able to access actual location of derived datasets and are required to obtain necessary information from the cache server instead. As such, all the usage history and statistics can be collated at this single point and Nectar will perform monitoring for the automatic garbage collection of derived datasets deemed to have the lease value. At the same time, it is necessary to remove cache entries which are not useful as datasets referenced by these entries cannot be deleted. The cache eviction policy includes a cost-benefit analysis to determine which entries to remove as follows: $CBRatio = (S \times \Delta T)/(N \times M)$, where S is the of the resulting derived dataset, ΔT is the elapsed time since it was last used, N is the number of times it has been used and M is the cumulative machine time of the computation that created it.

IV. DISCUSSIONS

This section will discuss MapReduce performance, handling stragglers, optimizing network topology, and optimizing data shuffle.

A. MapReduce Performance

The original MapReduce model itself is not without controversy. Michael Stonebraker et al. [8, 9] strongly argues with their benchmark results for MapReduce model against traditional parallel-RDBMS model that MapReduce does not offer significant better performance than the later. Although parallel-RDBMS model loads data slower than MapReduce model, the task was done faster in parallel-RDMBS model. It is observed that doing the task using traditional RDBMS, in addition to SQL, is much easier than writing user-defined functions for MapReduce, however MapReduce is easier to install and setup on parallel computing environment. The MapReduce model used in the benchmark is Apache implementation, Hadoop[5], however the authors believe that Google's MapReduce also suffers the same problem. Although this benchmark argues over the effectiveness of MapReduce model, the authors believe that in some applications, MapReduce is feasible to use [8]. Such applications may include complex analytics jobs (as Google uses MapReduce in order to index websites [10]) and quick-and-dirty analysis jobs (as MapReduce is easier and faster to install and configure).

B. Handling Stragglers

Google MapReduce has a general mechanism to handle stragglers by using backup tasks. Although this mechanism has been tuned to control the usage of additional computational resources and significantly reduces the job completion times of large MapReduce operations, there may be some problems such as backup task placement and unnecessary duplication and resources incurred. Mantri [13] is designed around such cons as it attempts to understand the causes of outlier and find out the best course of action to alleviate the issue of stragglers. The decision is based on resources available and opportunity costs considerations as Mantri identifies and acts on outliers early in order to release resources for usage by other task and speed up job completion time resulting in an improvement over MapReduce implementation that only duplicates the in the progress tasks near the end of operation. The performance of Mantri has been shown to be practically feasible where Mantri sped up the median job by 32% in the live deployment of BING in the production cluster and 55% of the jobs experienced a net reduction in resources used. The network-aware placement of tasks also speeds up half of the reduce phases by at least 60% each and completion times due to recomputation of jobs are reduced by at least 40% [13]. Nevertheless, an assumption is made where the cluster involved is homogenous where every available machine will have resources to perform the recomputation of a particular assigned task by Mantri. Therefore, a specialized cluster software will be necessary to manage the cluster of machines to enable load-balancing for Mantri's performance to be stipulated above.

C. Optimizing Network Topology

On the other hand, Camdoop optimizes MapReduce model over the network topology. We find that this is not really a breakthrough over the original MapReduce model, but is only a minor tweak of the network topology, although the tweak improves the runtime. Camdoop also makes use of CamCube platform, requires the current network topology to be re-setup. This might not a feasible workout for current environment, but is doable for fresh environments. Also, as CamCube is part of Microsoft Research project that is still ongoing, we suggest that CamCube is not used in a productive environment, but instead in a research environment to exploit the opportunities further and finalize the topology. Although the authors of Camdoop specify that their model is beneficial, the side effects are yet to be seen.

D. Optimizing Data Shuffle

The four model discussed in this report optimize the original MapReduce in different ways. Sudo optimizes on user-defined functions and data-shuffling stage. We think that this optimization is good that it works on low-level MapReduce model; although the authors of Sudo test the model based on SCOPE programming model and make

some adjustments, it is not difficult to implement Sudo model into existing MapReduce model using any high-level programming languages. The authors believe that Sudo will open many ways of further optimizations on parallel-computing world that not only learns the data from databases or distributed system, but also programming languages and system analysis. Therefore, we argue that Sudo is one of the feasible technique to develop optimal MapReduce model. However, Sudo model has some issues, that are discussed in the last section of this article.

E. Redundant Computations and Data Management

The MapReduce framework while simplifying large distribute data intensive applications, still lacks a scheme integrated to optimize data resources management. Hence Nectar aims to extend such functionality by avoiding redundant computations and removing datasets that are deemed to be not useful [5]. Nectar caches immediate results as well as programs that produced these derived results and hence cached results can be reused and if data that are removed and needed in the future, the programs can be rerun to provide for the results. Similarly, Nectar has been shown practical results through live deployment on 240-node research cluster as well as analytic results of execution logs from 25 large production clusters and on average across all clusters, more than 35% of the jobs benefit from caching [5]. However, Nectar only presents data management for derived datasets while primary datasets are not automatically deleted due to an inherent property of LINQ program [5]. As the operations scale accordingly, the condition of primary datasets growth which is not really taken care of in a comparable performance level relative to derived dataset may result in undesirable issues as well.

V. CHALLENGES

This section will discuss challenges on MapReduce as well as other improvement models already discussed in this article.

A. MapReduce

Since its publication, many researches and new products have been created in complement of MapReduce model. Google has developed and published Dremel in 2010 to further reduce the execution time of MapReduce by a fraction [7]. As data tends to get bigger and bigger and computation gets more complicated, it is interesting to see whether the optimizations described in this report are still applicable.

B. Sudo and Camdoop

The authors of Sudo mention out some issues that they call “interesting and somewhat negative” results. First, SCOPE programming model is different from the traditional MapReduce model; this model allows changing the key for reduce and merge tasks. They only study this property on SCOPE model, thus we think there is still room for optimization for studies that exhibit similar optimization but on a more general environment (i.e. the MapReduce model itself). Second, for quite a few jobs, loading input data incurs the most I/O cost. This is due to loading more data than necessary. Since the occurrence are very small, we think that this is not an issue. Third, for some jobs, the first shuffling-phase dominates the shuffling cost. This is due to the fact that the output is significantly smaller in number than the input. The authors mention a pipeline opportunity to be sought. Fourth, the rule-based deduction can be improved by making the analysis context- and path-sensitive. Nevertheless, this model gives quite an improvement to the original MapReduce model. If it is applicable to any high-level programming language, distributed systems will gain more benefit from it.

Meanwhile, the authors of Camdoop have tested that Camdoop over CamCube topology has a very significant improved performance over traditional MapReduce models, such as Apache Hadoop and Dryad (now known as LINQ to HPC [4]). They show that in every case, even with small input data, Camdoop over CamCube outperforms both platforms in terms of shuffle and reduce times. However, there are newer versions of these two platforms (at the time of writing, there are multiple newer versions of Apache Hadoop [5] and Microsoft now focuses to bringing Apache Hadoop into Windows Servers and Azure [4, 6]), so a reevaluation might be needed to see whether the Camdoop optimization over traditional MapReduce still holds. The authors also note out that currently Camdoop is not open for some optimizations on traditional MapReduce model, such as support for iterative jobs, incremental computations, and pipelining of Map or Reduce phase. However, they believe that these optimizations are quite non-trivial and may be incorporated into Camdoop model in the future.

C. Mantri and Nectar

Based on the way that Mantri is structured, it can be seen as another layer over MapReduce which performs this detection and handling of stragglers. Perhaps, with such enhanced understanding of the stragglers problem, the next step can to deviate from enforcing a layer of “policy” and to change some inherent properties of MapReduce to better mitigate the problem of stragglers across production clusters.

A common observation is that the caching mechanism of Nectar is dependent on the nature of the computation and hence an assumption made would be that the results of DryadLINQ applications are deterministic. Therefore, computations do not produce the same result all the time may cause a failure of the caching operations. This will definitely introduce increased complexity should the algorithm be modified and improved upon to cater for such computations.

D. *Real Time Woes*

MapReduce excels when general query mechanism is carried out on large volume of data. However, it will fail to perform as well on applications which require real-time processing [15]. As described in section 2, MapReduce programming model involved a Map pre-processing step and a Reduce data aggregation step. While Map step can be applied on real-time streaming data, Reduce step may not be able to work properly as all input data for each unique data key are required to be mapped and collated first. Although there are techniques to overcome this limitation, there will be a new set of problems brought forth to the table.

VI. CONCLUSION

In this article, we have discussed MapReduce as a data processing tool used in Cloud Computing for the big data involved in applications of today. MapReduce is no doubt an excellent tool which provides great scalability and fault tolerance with simplicity. However, it does come with a set of limitations in its framework, and we have studied in details some optimization and extensions to overcome these limitations as presented in recent approaches. Nonetheless, challenges and improvements remain for these enhanced models, and the paradigm that MapReduce governed by more new and upcoming complicated policies could possibly a step backwards remains an interesting one.

VII. REFERENCES

[1] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. Comm. of ACM 51, 1 (2008).
[2] Paolo Costa, Austin Donnelly, Ant Rowstron, Greg O'Shea. "Camdoop: Exploiting In-network Aggregation for Big Data Applications". Proceedings NSDI, April, 2012
[3] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y.Li, Wei Lin, Jingren Zhou, and Lidong Zhou, "Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions", in Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12), USENIX, 25 April 2012.
[4] MSDN: LINQ to HPC. http://msdn.microsoft.com/en-us/library/hh378101.aspx. Retrieved 12 November 2012.
[5] Apache Hadoop Release Notes. http://hadoop.apache.org/releases.html#News. Retrieved 12 November 2012.
[6] Foley, Mary Jo. Microsoft to develop Hadoop distributions for Windows Server and Azure. ZDNet. http://www.zdnet.com/blog/microsoft/microsoft-to-develop-hadoop-distributions-for-windows-server-and-azure/10958. Retrieved 12 November 2012.
[7] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, "Dremel: Interactive Analysis of Web-Scale Datasets", in Proceedings of the 36th International Conference on Very Large Data Bases (2010), pp. 330-339.
[8] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and Parallel DBMSs: Friends or Foes?," Communications of the ACM, vol. 53, iss. 1, pp. 64-71, 2010.
[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data, New York, NY, USA, 2009, pp. 165-178.
[10] David F. Carr. How Google Works. http://www.baselinemag.com/c/a/Infrastructure/How-Google-Works-1/. Retrieved 12 November 2012.
[11]. Ant Rowstron. Rethinking the Data Center: CamCube and beyond. http://research.microsoft.com/en-us/um/people/antr/borgcube/borgcube.htm. Retrieved 12 November 2012.
[12] Rimal. P, E. Choi, and I. Lan, "A Taxonomy and Survey of Cloud Computing Systems". 2009 Fifth International Joint Conference on INC, IMS and IDC
[13] A. Ganesh, S. Kandula, and A. Greenberg, "Reining in the Outliers in Map-Reduce Clusters using Mantri", 9th USENIX Symposium on Operating Systems Design and Implementation
[14] P. Gunda, L. Ravin, C. A. Thekkath, Y. Yu, L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters"
[15] Google's Colossus Makes Search Real-time by Dumping MapReduce. http://highscalability.com/blog/2010/9/11/googles-colossus-makes-search-real-time-by-dumping-mapreduce.html. Retrieved 12 November 2012.